Finite State Machine

References

Lecture 06
Yinong Chen
Roadmap:
Evaluation in Design Process

1. Combinational and Sequential Circuits
2. Stateless Vending Machine Design
3. Finite State Machine
4. Examples of Finite State Machines
Combinational Circuits

- Stateless: information cannot be stored in the circuit;
- Output is determined by input only;
- Truth table fully specifies the function

<table>
<thead>
<tr>
<th></th>
<th>a</th>
<th>b</th>
<th>CarryIn</th>
<th>CarryOut</th>
<th>Sum</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>
Sequential Circuits

- The circuit stores state (internal values calculated in the past)
- Output is determined by input and state;
- Finite state machine specifies the function.

Example: an accumulator

Diagram:
- Adder
- Running sum
- State
Combinational and Sequential Circuits

Combinational Circuit

Combinational Circuit

State / memory
Finite State Machine (FSM)

- Truth table serves as the specification of
  - Combinational circuit (hardware)

- An Finite State Machine serves as the specification of
  - a sequential circuit (hardware), to be taught in CSE 120, and
  - **an event-driven program (software)**
Model  For a Stateless Vending Machine

For a given input, it gives an output immediately

- Problem Definition: Use these US currency coins to purchase products in the machine;
- Parameters: coins and products
- Range of values for each parameter:
  - Coins: 1, 5, 10, 25
  - Products: sicker, candy, pencil, and marker
- Constraints/Relationships /Solution (function table):

<table>
<thead>
<tr>
<th>Coins</th>
<th>Penny (1)</th>
<th>Nickel (5)</th>
<th>Dime (10)</th>
<th>Quarter</th>
</tr>
</thead>
<tbody>
<tr>
<td>Products</td>
<td>Sicker</td>
<td>candy</td>
<td>pencil</td>
<td>marker</td>
</tr>
</tbody>
</table>
A Stateless Vending Machine

VPL Implementation: No variable is used
Why does a coin-operated washing machine take all coins at the same time?
A four-way intersection has red/green traffic lights that are controlled with timers.

Traffic can only move in one direction at a time: NS (North-South) or EW (East-West).
A Finite-State Machine (FSM) is a model of the discrete dynamics of a system that has a finite number of discrete states. Transitions between states are caused by events, such as:

- the expiration of a timer
- a change in a sensor value

**State Diagram**

- EW green to EW red (timer expired or emergency signal)
- NS red to NS green (timer expired or emergency signal)

**State Table**

<table>
<thead>
<tr>
<th>current state</th>
<th>event</th>
<th>next state</th>
</tr>
</thead>
<tbody>
<tr>
<td>EW grn/NS rd</td>
<td>timer exp</td>
<td>EW rd/NS grn</td>
</tr>
<tr>
<td>EW rd/NS grn</td>
<td>timer exp</td>
<td>EW grn/NS rd</td>
</tr>
</tbody>
</table>
The Traffic Lights by Canary Wharf Tower, East London
A Finite State Machine is a mathematical model consisting of a finite number of states, transitions between states, inputs, and outputs.

Finite State Machines are designed to respond to a sequence of inputs (events), such as
- coin insertions into a vending machine
- mouse-clicks/key strikes during a program’s execution
- The arrival of individual characters from a string

Each input causes a transition from one to another state

An output can be associated to an input
Finite-State Machines are often used to design control systems...

When button pressed:
If state==open
then    close
else    open

No states required

Required states
Finite-State Machines are often used to design control systems...

A garage door opening system

If the door is closed and I press the button (touch sensor), the door begins to move up.

When it reaches the top, the door activates a limit switch (a touch sensor) and stops.

If the door is open and I press the button, the door begins to move down.

When it reaches the bottom, the door activates another limit switch and stops.
Finite-State Machines are often used to design control systems...

A garage door opening system

...we want to design the controller...
Finite-State Machines are often used to design control systems...

A garage door opening system

states
- door closed
- door open
- door closing
- door opening

events
- button press
- limit switch touched
  (closing finished or opening finished)
Finite-State Machines are often used to design control systems...
Finite-State Machines are often used to design control systems...

Diagram:
- **Closing Stopped**
- **Door Closed**
- **Door Opening**
- **Opening Stopped**
- **Door Closing**
- **Door Opened**

Transitions:
- Button pressed from **Door Closed** to **Door Opening**
- Button pressed from **Door Opening** to **Opening Stopped**
- Button pressed from **Door Closed** to **Door Closing**
- Button pressed from **Door Closing** to **Door Opened**
- Limit tripped from **Door Closed** to **Closing Stopped**
- Limit tripped from **Door Opened** to **Door Closing**
ASU-VPL Implementation if the Garage Door Opener
Example 1: Detecting Even or Odd

- The following FSM determines whether the number of 1s is even or odd, for a given binary number, e.g., 1001010110
  - Circles represent states; arrows represent transitions
  - Input is binary number or a string 0s and 1s
  - The “output” indicates the current state
Example 2: Nested Parenthesis

The following example tests whether parentheses are properly nested (up to 3 deep)

\[(x*(y-z)+2*(y+(x-3*z)))\]

- start
- OK
- 1
- 2
- 3
- Error
- "mismatch"

\(\nabla: \) anything but "(" and ")"
Nested Parentheses
Using an Additional Variable

If input = "(" \rightarrow count++

If input ")" AND count == 1
\rightarrow count--

If input "(" \rightarrow count=1

If input ")" AND count >1
\rightarrow count--
Example 3: FSM Vending Machine

- Takes quarters and dollars only
- Maximum deposit is $1 (or four quarters)
- Sodas cost $0.75

Possible Inputs (Events):
- Deposit quarter (25)
- Deposit dollar (100)
- Push button to get soda (soda)
- Push button to get money returned (ret)

States: 0, 25, 50, 75, 100, and state transits on input
Example 3: FSM Vending Machine

Deposit quarter
Deposit dollar
Push button to get soda (soda)
Push button to return money (ret)
If (Sum == 75)
release soda

If (Sum > 75)
Sum = Sum - 75
release soda
If Sum < 75, do nothing

Sum = Sum + 25
Sum = Sum + 100

soda
dollar
quarter
return
Example 3: FSM Vending Machine
An autonomous mobile robot must navigate through a maze.

An on-line navigation problem: solving a maze from the inside.

An on-line algorithm receives its input gradually rather than all at once.

It must make decisions based on this partial input.
Online Programming of Wall Following Robot

http://venus.eas.asu.edu/WSRepository/eRobotic/
VPL Implementation

- Install ASU Maze into:
  
  C/Document and Settings/User/Microsoft Robotic Dev Studio 4/samples/Config
FSM for Hard-Coded Turns using Touch Sensor

FW 1
Turning Left 90
Touched

FW 2
Turning Left 1
Touched
Left Finished

FW 3
Turning Right 90
Touched

FW 4
Turning Right 1
Touched
Right Finished

FW 5

Backward before turning

Turned Right 2
Left Finished

Turned Left 1

Turned Left 2
Left Finished

Backward before turning
FSM of Robot in a Maze

Start

- Forward
  - DistanceMeasured < 400
  - Resum180 Finished
    - RightDistance >= rightDistance
      - Resume 180
    - DistanceMeasured < rightDistance
      - Turned Left

- Turning Right
  - rightFinished
    - Turned Right
    - RightDistance ← DistanceMeasured

- Turning Left
  - LeftFinished
    - Turned Left
Implementation of the ‘right-then-left’ FSM

Intel Edison-based robot with built-in Wi-Fi and Bluetooth components and a distance sensor.
Greedy Algorithm based on the First Working Solution

- **Start**
  - **Forward**
    - DistanceMeasured < 400
      - **Turning Left**
        - leftFinished
    - Spin180
      - Finished
  - Spin180
    - Finished
  - DistanceMeasured >= 800
    - **Turned Left**
      - DistanceMeasured < 800
        - **Turned Left**
Right-Wall-Following Algorithm with Error Correcting

- **Forward**
  - DistanceMeasured < BaseDistance - 5
  - Touch-Sensor Touched
  - DistanceMeasured > BaseDistance + 5

- **DistanceMeasured > BaseDistance + 400**
  - Turned Right
  - Turned Left
  - Turning Left90
  - Turning Right90

- **Touch-Sensor Touched**
  - Left 1 degree
  - Right 1 degree

- **Backward then turn**
  - Forward a bit then turn

- **Start**
  - leftFinished
  - rightFinished
Wall-Following Diagram in VPL

- **Init**
  - Comment: The init block will initialize variables and start the robot moving forward.

- **Lego EV3 Ultrasonic**
  - **If**
    - `state.Status == "Forward" AND value < state.BaseDistance - 5`
    - `state.Status == "Forward" AND value > state.BaseDistance + 50`
    - `state.Status == "Forward" AND value > state.BaseDistance + 5`

- **Lego EV3 Touch**
  - **While**
    - `state.Status != "Forward"`

- **Data**
  - TurningLeft1
  - TurningRight90
  - TurningRight1
  - Status

- **Variable**
  - Status

- **End While**

- **Data**
  - TurningLeft
  - String

- **Variable**
  - Status

- **End While**

- **Data**
  - TurningLeft1
  - String

- **Variable**
  - Status

- **End While**

- **Data**
  - TurningRight90
  - String

- **Variable**
  - Status

- **End While**

- **Data**
  - TurningRight1
  - String

- **Variable**
  - Status

- **End While**

- **Data**
  - TurningLeft
  - String

- **Variable**
  - Status

- **End While**

- **Data**
  - TurningLeft1
  - String

- **Variable**
  - Status

- **End While**

- **Data**
  - TurningRight90
  - String

- **Variable**
  - Status

- **End While**

- **Data**
  - TurningRight1
  - String

- **Variable**
  - Status

- **End While**

- **Data**
  - TurningLeft
  - String

- **Variable**
  - Status

- **End While**
1. You have two minutes to run the course forward and return; You can score up to 20 + bonus points, as shown on the map;
2. The farthest position is used for calculating the score;
3. If forwarding failed in the middle, you can take the robot to the end position to run the backward part;
4. If you use sensor to detect the front wall, + 10% bonus
5. If you use sensor(s) to detect front and side walls + 20% bonus points;
6. If you do not touch robot for the return trip, you receive 2 bonus points.